SUPER COMPUTER

Contributions

Performance of various computers using standard techniques for solving sparse linear equations

Jack J. Dongarra

The University of Tennessee, Computer Science Department, 104 Ayres Hall, Knoxville TN 37996-1301, USA
and
Oak Ridge National Laboratory, Mathematical Sciences Section, PO Box 2008, Bldg 6012, Oak Ridge TN 37831-6367, USA

Henk A. van der Vorst

University of Utrecht, Mathematical Institute, PO Box 80 010, NL-3508 TA Utrecht, The Netherlands

Supercomputer 51, IX–5
Received May 1992

The benchmark program presented here is a first attempt to construct a representative test problem for sparse matrices on a variety of machines. The heart of the code is a conjugate gradient (CG) iterative solver for large, sparse, symmetric and positive definite linear systems. The performance range and difficulties encountered on a specific architecture, made visible by this simplified model problem, are more or less representative for a class of iterative method (Lanczos, Bi-CG, CGS, etc.; see [1] for further details).

1. Introduction and motivation

The LINPACK benchmark [2] has become popular in the past few years as a means of measuring floating-point performance on computers. The benchmark shows in a simple and direct way the performance to be expected for a range of machines when doing dense matrix computations. Many people have discovered that the situation for *sparse* matrices is less clear. For very large, structured sparse systems, if one applies a dense direct solver operating on a banded matrix, one may expect computing times similar to that of the dense case. In other situations, however, the performance strongly depends on the non-zero structure of the matrix, the storage scheme used, and the algorithm chosen.

The algorithm used in the LINPACK benchmark is a direct method. That is, the original matrix is factored into the product of two simpler matrices, a lower triangular matrix L and an upper triangular matrix U, and these matrices are used to construct the solution. The factorization requires $O(n^3)$ operations and the solution $O(n^2)$. Thus, the solution can be determined in a fixed number of operations. A direct method can be impractical, however, if the matrix is large and sparse, because the sought-after factors can be dense and the time to construct the factors large.

Many important practical problems give rise to large sparse systems of linear equations. One reason for the great interest in sparse linear equations solvers and iterative methods is the importance of being able to obtain numerical solutions to partial differential equations. Such systems appear in studies of electrical networks; models of economic systems;

This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, US Department of Energy, under Contract DE-AC05-84OR21400, and in part by the Stichting Nationale Computer Faciliteit (NCF), The Netherlands.

and physical processes such as diffusion, radiation, and elasticity. Iterative methods work by continually refining an initial approximate solution so that it comes closer and closer to the correct solution. With an iterative method a sequence of approximate solutions $\{x^{(k)}\}$ is constructed which essentially involve the matrix A only in the context of matrix-vector multiplication. Thus the sparsity can be exploited so that each iteration requires $O(n)$ operations.

As a result of the algorithm, sparsity of the matrix, and the data structures, the time-consuming elements of iterative methods can be quite different from the time-consuming elements of a direct solver. Thus, the performance of an iterative method can be completely different from the performance of a direct solver.

The purpose of this report is to describe the performance of various computers when executing a Fortran implementation of common algorithms used in solving sparse matrix problems.

We are not interested in comparing iterative schemes, or in measuring the number of iterations needed to converge, or even in obtaining the solution. Our intent here is to provide a means for quickly testing and comparing the performance of a computer on commonly occurring solution methods for solving sparse matrix problems.

In Section 2 we provide an overview of the conjugate gradient algorithm and give pointers to related approaches. Section 3 describes the test problem, the structure of the matrix used in this evaluation, and the preconditioners used in the algorithm. Section 4 describes the tests and displays the performance for the various machines and the approaches used. In addition, instructions for running the test are provided, as well as a description on how the software can be obtained. Section 5 presents some conclusions.

2. Conjugate gradient algorithm and related algorithms

We begin with a discussion of the classical conjugate gradient algorithm. The classical conjugate gradient algorithm for the solution of $Ax = b$ can be represented by the following scheme:

x_0= initial guess; $r_0 = b - Ax_0$
$p_{-1} = 0; \beta_{-1} = 0$
$\rho_0 = (r_0, r_0)$
for $i = 0, 1, 2, \ldots$
  $p_i = r_i + \beta_{i-1} p_{i-1}$
  $q_i = Ap_i$
  $\alpha_i = \frac{\rho_i}{(p_i, q_i)}$
  $x_{i+1} = x_i + \alpha_i p_i$
  $r_{i+1} = r_i - \alpha_i q_i$
  if x_{i+1} accurate enough then quit
  $\rho_{i+1} = (r_{i+1}, r_{i+1})$
  $\beta_i = \frac{\rho_{i+1}}{\rho_i}$
end

In this algorithm the variables α, β, and ρ are scalars, x, r, p, and q are vectors of length n, $(x, y) = x^T y$, and A is a sparse n by n symmetric positive definite matrix.

The speed of convergence of the conjugate gradient method depends strongly on the spectrum of the matrix in the system. It is often advantageous to select a matrix, K, as a preconditioner for the system that clusters the eigenvalues of the system. The following computational scheme is for preconditioned CG, for the solution of $Ax = b$ with preconditioner K^{-1}.

x_0= initial guess; $r_0 = b - Ax_0$
$p_{-1} = 0; \beta_{-1} = 0$
Solve w_0 from $Kw_0 = r_0$
$\rho_0 = (r_0, w_0)$
for $i = 0, 1, 2, \ldots$.
  $p_i = w_i + \beta_{i-1} p_{i-1}$
  $q_i = Ap_i$
  $\alpha_i = \frac{\rho_i}{(p_i, q_i)}$
  $x_{i+1} = x_i + \alpha_i p_i$
  $r_{i+1} = r_i - \alpha_i q_i$
  if x_{i+1} accurate enough then quit
  Solve w_{i+1} from $Kw_{i+1} = r_{i+1}$
  $\rho_{i+1} = (r_{i+1}, w_{i+1})$
  $\beta_i = \frac{\rho_{i+1}}{\rho_i}$
end

An alternative would have been to transform the system explicitly to the symmetric system with matrix $L^{-1}AL^{-T}y = L^{-1}b$, but then the solution needs to be backtransformed to $x = L^{-T}y$ afterwards. The above formulation has the advantage that the preconditioner need not be split into two factors. Further, it avoids backtransform solutions and residuals, as is necessary when one applies classical CG to $L^{-1}AL^{-T}y = L^{-1}b$ (with $K = LL^T$).

Performance of other iterative solvers. The Mflop rate per iteration step of CG is almost entirely determined by the performance of the matrix-vector product (Ap_i) and the preconditioner (solve w_{i+1} from $Kw_{i+1} = r_{i+1}$). In biconjugate-gradient (Bi-CG) methods, the computational work per iteration step is almost double the work for conjugate-gradient (CG) methods; each pass of the iteration loop now involves a matrix-vector product with A and A^T (and the preconditioned version involves solve steps with K and K^T). In the case that the steps with transpose operators can be computed with similar Mflop rates as for the original operators, one may expect the same Mflop rate for Bi-CG as for CG. In many situations, however, one has a data structure which facilitates the efficient computation of products like Ap_i (e.g., row-wise compressed), and then a product like $A^T\tilde{p}_i$ often requires indirect addressing. This, of course, may lead to a reduced Mflop rate for Bi-CG in comparison with CG.

For the quasi-minimum residual (QMR) alorithm [3], in which the bulk of the computational work is the same as for Bi-CG, a similar situation holds. In other variants on Bi-CG the computation of $A^T\tilde{p}_i$ has been avoided and has been replaced by a product involving A (also, the solve with K^T has been replaced by a solve with K). The more well-known of these schemes are CGS, Bi-CGSTAB, and TFQMR. These schemes lead to almost identical Mflop rates as those for CG.

For iterative schemes that are based on the generation of a full orthogonal basis for the search space (e.g., GMRES, ORTHODIR, ORTHORES, ORTHOMIN, and GENCG), the situation is more complicated. Typically, in the full versions (i.e., not truncated or restarted) of these schemes, we see that in the i-th iteration step one matrix-vector product with A, one solve with K (in the preconditioned versions), i inner-products, and i vector updates are involved. For low values of i we therefore may expect similar Mflop rates as for CG, while for increasing values of i the work in the inner-products and the updates tends to dominate, and hence the Mflop rate is increasingly determined by these operations.

3. Test problem and conjugate gradient solver

For this benchmark, the test problem is generated by discretizing a 3-D elliptic partial differential equation by the standard 7-point central difference scheme over a 3-D rectangular grid, so that we have 100 unknowns in each direction. This procedure results in a system of order 1,000,000. The system is scaled such that it has a unit diagonal (which is common practice for many iterative schemes). For our test problem, the CG method iterates on a well-structured 7-diagonal model problem:

$$A = \begin{pmatrix} x & x & & x & & & x & & & \\ x & x & x & & x & & & z & & \\ & x & x & x & & x & & & z & \\ & & x & x & x & & z & & & \ddots \\ & & & x & x & & & z & & \\ x & & & & x & x & & & \ddots & \\ & x & & & & \ddots & \ddots & & & \\ & & \ddots & & & & & & & \\ x & & & & & & & & & \\ & x & & & & & & & & \\ & & \ddots & & & & & & & \end{pmatrix}$$

There is even more structure in the matrix A. The matrix A can be seen as a symmetric block tridiagonal matrix:

$$A = \begin{pmatrix} A_1 & D_1 & & \\ D_1 & A_2 & D_2 & \\ & D_2 & \ddots & \ddots \\ & & \ddots & \end{pmatrix},$$

in which the D_i are diagonal matrices, and the A_i are again symmetric block tridiagonal matrices:

$$A_i = \begin{pmatrix} A_{i,1} & C_{i,1} & & \\ C_{i,1} & A_{i,2} & C_{i,2} & \\ & C_{i,2} & \ddots & \ddots \\ & & \ddots & \end{pmatrix}.$$

The $C_{i,j}$ are diagonal matrices, and the $A_{i,j}$ are symmetric tridiagonal matrices.

Admittedly, the generated problem might be solved more efficiently by other methods (see [3] for a survey of methods). We have tried to avoid comparing iterative schemes. In fact, we are not interested even in the solution of the system. We ignore properties of the given system, other than its symmetry and its non-zero diagonal structure. Our objective is to obtain a quick impression of the performance of a generic sparse matrix solver (see the Appendix for our examples).

We have also avoided discussion of suitable stopping criteria. Thus, we simply carry out 50 iteration steps of the CG method (we have implemented an inexpensive test to avoid excessive iteration steps if one solves smaller systems). These 50 iteration steps allow for a sufficiently accurate CPU timing, and they also enable some realistic optimization of the algorithm – for example, by grouping iteration steps in order to reduce data transfer (see [1]). Moreover, this number is representative of many actual situations and is not too large that running the code is unnecessarily expensive.

Conjugate gradient schemes. We report performance rates for three different solution schemes: scaled CG (i.e., with diagonal scaling of the matrix), standard ICCG, and ICCG with diagonal ordering. Of these three solution schemes scaled CG will often require the most iterations to converge. On the other hand, it is easily vectorizable and parallelizable, and it is cheaper in terms of floating-point operations per iteration step.

Since CG is usually combined with some kind of preconditioning, we have also carried out 50 iteration steps of CG with standard incomplete

Cholesky factorization as a preconditioner. The CPU time for the construction of that preconditioner has not been included in our timing.

The performances for these (preconditioned) CG algorithms are thought to be representative of a class of well-structured problem. For less well-structured problems, the performance for scaled CG is more or less determined by the performance for the sparse matrix-vector product (usually determined by indirect addressing). In such cases, the standard lexicographical ordering preconditioned CG gives a reasonable impression of the performance of preconditioned CG. Diagonal ordering is not feasible for more general non-zero structures (though in some cases some improvement can be obtained by reordering the unknowns; see Radicati *et al* [4]).

The number of operations for the scaled CG method is about $22n$ flops per iteration for our model problem, whereas the preconditioned versions take $35n$ flops per iteration. (We count both additions and multiplications as distinct floating-point operations.)

In judging the performance of the algorithms, it is important to take into account not only the execution rate for the methods, but also the number of operations per iteration and the number of iterations required to converge. As an example, in Figure 1 we see that CG with diagonal scaling has the highest execution rate; however, since more iterations usually will be required for convergence, the overall solution time may be longer.

Other preconditioners. A variety of preconditioners are suggested in the literature. The Mflop rates for these preconditioners may vary widely (as well as their effect on the number of iteration steps). Some of these preconditioners are specifically designed for vector computers (e.g., those based on truncated Neumann series) or for parallel computers (e.g., the preconditioners constructed on overlapping domains). Also, some incomplete block decompositions lend themselves more to vector computers, when applied to matrices with regular sparsity patterns. The success of these block decompositions seems more apparent for 2-D problems; for 3-D problems their effect seems to be less significant with respect to point incomplete decompositions.

Another approach is to apply the preconditioners in combination with some coloring scheme, for example, red-black ordering. While this may lead to higher Mflop rates, it often also leads to significant increases in the number of iteration steps, so that CPU time or wall clock time may not decrease. For more complicated coloring schemes (from 20 to 100 colors), the number of iteration steps is reported to increase only marginally at most.

Finally, one may apply preconditioning with a polynomial in A itself. Needless to say, in this case the Mflop rate of the preconditioned iterative scheme is largely determined by the Mflop rate for the matrix-vector product.

4. Results

In Figure 1 we show the results for the various machines. The linear system is generated by discretizing a 3-D, second-order, elliptic partial differential equation by finite differences over a rectangular equidistant mesh, with Dirichlet boundary conditions. We do not specify the PDE here, since the performance of the method should not depend on knowledge of the PDE: one is not allowed to take any advantage of the PDE other than the structure of the resulting system.

The four diagonals of the upper triangular part of the symmetric matrix that contain non-zero elements are stored in the first four columns of an array `A( *, 10 )`.

`A( i, 1 )` the element on the main diagonal (on the i-th row),

`A( i, 2 )` the element on the first codiagonal (on the i-th row),

`A( i, 3 )` the element on the $(nx+1)$-th codiagonal (i-th row, $nx = 100$),

`A( i, 4 )` the element on the $(nxny+1)$-th codiagonal (i-th row, $nxny = 10,000$).

The 5th column of `A` is used to store the right-hand side of the linear system, the 6th column contains the diagonal of the preconditioner, columns 7 through 9 are used as working space for CG, and the 10th column contains scaling information for the matrix A (this is necessary for backscaling the solution).

For unpreconditioned CG the matrix A has been scaled so that `A( i, 1 ) = 1` for all i. This property has been exploited in the computation of the matrix-vector product. The incomplete Cholesky preconditioning matrix K (for preconditioned CG) can be written as

$$K = (L + \hat{D})\hat{D}^{-1}(\hat{D} + L^{\mathrm{T}}),$$

in which L^{T} is the strictly upper triangular part of the matrix A. In this case the matrix A has been scaled symmetrically in such a way that $\hat{D} = I$. This is, of course, exploited in the preconditioning part. In this case, the $a(i,1)$ are not necessarily equal to 1.

The block structure of A has not been exploited in our code; the block structure of K has been exploited in the "diagonalwise ordering" variant. The parts of the code for the computation of the matrix-vector product (along with a required inner-product for CG) and the preconditioning part are given in the Appendix.

The information in Figure 1 was compiled over a period of several months. Subsequent systems software and hardware changes may alter the timings to some extent.

One further note: In multiprogramming environments it is often difficult to reliably measure the execution time of a single program. Anyone evaluating machines and operating systems should gather additional data. The column labeled "Standard ICCG" refers to the use of the standard lexicographical ordering of the unknowns. This approach leads to recurrence relations. These relations can be partly vectorized by loop splitting.

Machine	Standard ICCG	Diag-order ICCG	Scaled CG	Peak performance
NEC SX-3/22 (2.9 ns, 1 proc.)	60.4	607	1124	2750
Cray Y-MP C90 (4.2 ns, 1 proc.)	56.0	444	737	952
IBM 9000 Model 820 (1 proc.)	43.5	39.6	74.6	444
Cray Y-MP/464 (6 ns, 1 proc.)	38.0	175	261	333
Cray 2 (4.1 ns, 1 proc.)	27.5	96.0	149	500
IBM 3090 Model S (1 proc.)	19.1	13.5	32.3	133
IBM RS/6000-550 (24 ns)	18.8	18.3	21.1	81
IBM 9121 (15 ns) (1 proc.)	14.1	10.6	25.4	133
DEC Vax/9000 (16 ns, 1 proc.)	10.2	9.48	17.1	125
Convex C3210 (1 proc.)	6.65	15.8	19.1	50
Convex C-240 (1 proc.)	6.45	11.7	13.8	50
Alliant FX/2800 (1 proc.)	2.66	2.18	2.98	40
DEC Vaxstation 5000/200	2.57	2.24	2.84	
DEC Vaxstation 5000/125	1.49	1.33	1.58	

Figure 1. Speed in Mflop/s for 50 iterations of the iterative techniques.

This has not been coded explicitly in the body of the preconditioner (see the Appendix routine `prec3s`), but we have included an example in this document (Appendix routine `prec3p`).

The column labeled "Diag-order ICCG" refers to the use of diagonal ordering of the unknowns over the x, y planes of the grid [5], [1]. The compiler should be instructed that there are no recurrences in this case (by means of compiler directives; see the Appendix rout ine `prec3d`). Diagonal ordering leads to vector code without indirect addressing, but with only modest vector lengths (about 50, on the average). Longer vector lengths can be obtained by the so-called hyperplane ordering (see [1]).

Obtaining the software. The software used to generate the data for this report can be obtained by sending electronic mail to `netlib@ornl.gov` .

To receive the single-precision software for this benchmark in the mail message to `netlib@ornl.gov` and type *send sparses from benchmark* .

To receive the double-precision software for this benchmark, type *send sparsed from benchmark* .

Rules for running the tests. The software intentionally has been kept simple so that it will be easy for a programmer to adapt the program, or parts of it, to a specific architecture with only a modest effort. In running the tests, the user is allowed to reorganize the CG algorithm or parts of it (e.g., grouping updates or combining inner-products with other operations); calls to the BLAS may be replaced by optimized code, and compiler optimization directives may be added. However, the user may not change the basic algorithm.

5. Conclusions

The state of optimization for sparse matrix algorithms is much different

from that of dense problems. This fact can be readily verified from the results in Figure 1. For dense problems one can typically exploit features of the architecture by using block versions of the algorithm (for cache and distributed memory), by combining successive updates (vector register machines), or by exploiting both methods. In fact, many dense linear algebra algorithms can be organized so that $O(n)$ arithmetic operations can be done per data transfer from main memory to the fastest memory. In contrast, the ratio of computational work to data transfer is usually rather low ($O(1)$) for iterative sparse matrix solvers, and the vectors are usually much too long to be kept in any local memory. Therefore, expected performance really depends on the architecture and the compiler. If the memory bandwidth is large, as for the Cray Y-MP, we may expect close to peak performance. Smaller bandwidths are almost immediately and inevitably reflected by smaller fractions of the peak performance. It then depends on the compiler how much advantage can be taken from a possible combination of operations in order to reduce the necessity of data transfer.

References

1
Dongarra, J.J., I.S. Duff, D.C. Sorensen and H.A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, 1991.

2
Dongarra, J. J., *Performance of various computers using standard linear equations software in a Fortran environment*, Computer Science Technical Report CS-89-85, University of Tennessee, March 1990.

3
Freund, R.W., G.H. Golub and N.M. Nachtigal, *Iterative solution of linear systems*, Acta Numerica 1, 57–100, 1992.

4
Radicati di Brozolo, G. and M. Vitaletti, *Sparse matrix-vector product and storage representations on the IBM 3090 with Vector Facility*, Technical Report 513-4098, IBM-ECSEC, Rome, July 1986.

5
Vorst, H.A. van der, *High-performance preconditioning*, SIAM J. Sci. Statist. Comput. 10, 1174–1185, 1989.

Appendix: details of the relevant parts of the software

A. Standard ICCG

The standard incomplete Cholesky preconditioning is carried out by subroutine `prec3s`.

```
      subroutine prec3s( x, px, n, nxny, nx, zkr)
      implicit double precision (a-h,o-z)
      dimension x(n), px(n)
      common /matrix/ a(1061208,10)
*
* Standard preconditioning ICCG(0)
*
      px(1) = x(1)
      do 10 i = 2, nx
        px(i) = x(i) - a(i-1,2) * px(i-1)
   10 continue
      nxl = nx + 1
      do 20 i= nxl, nxny
        px(i) = x(i) - a(i-1,2) * px(i-1) - a(i-nx,3) * px(i-nx)
   20 continue
      nxnyl = nxny + 1
      do 25 i = nxnyl, n
        px(i) = x(i) - a(i-1,2) * px(i-1) - a(i-nx,3) * px(i-nx) - a(i-nxny,4) *
     &       px(i-nxny)
   25 continue
*
* Lower triangular part has been solved
*
* Computation of (r,w), where w is the preconditioned residual
*
      zkr = ddot( n, px, 1, px, 1 )
      ij = n - 1
      do 30 i = 2, nx
        px(ij) = px(ij) - a(ij,2) * px(ij+1)
        ij = ij - 1
   30 continue
      do 40 i = nxl, nxny
         px(ij) = px(ij) - (a(ij,2) * px(ij+1) + a(ij,3) * px(ij+nx))
        ij = ij - 1
   40 continue
      do 50 i = nxnyl, n
        px(ij) = px(ij) - (a(ij,2) * px(ij+1) + a(ij,3) * px(ij+nx) + a(ij,4) *
     &       px(ij+nxny))
        ij=ij-1
   50 continue
*
* Diagonal and upper triangular part has been solved
```

```
*
      return
      end
```

B. Diagonal-order ICCG

For the 7-point stencil, the incomplete Cholesky preconditioner can be vectorized by reordering the unknowns diagonalwise over x, y planes of the grid. The subroutine `prec3d` carries out the preconditioning step over the whole grid and leaves the parts corresponding to x, y planes to the subroutine `precd`. These codes contain comment directives (in our example for Convex compilers) that instruct the compiler to ignore the apparent recurrence. These directives should be adjusted to the respective computer.

```
      subroutine prec3d( x, px, n, nx, ny, nz, rkr )
      implicit double precision (a-h,o-z)
      dimension x(n), px(n)
      common /matrix/ a(1061208,10)
      logical left
*
* Preconditioning for ICCG
* Vectorization by diagonalwise ordering of computations
* in (x,y) plane ([2]: Chapter 7.3.4)
*
      nxny = nx * ny
*
* Solving the lower triangular part of the preconditioner
*
      left= .true.
      do 10 i = 1, nxny
        px(i) = x(i)
   10 continue
      call precd( px(1), nxny, a(1,2), a(1,3), nx, left )
      do 30 j = 2, nz
        jl = (j - 1) * nxny + 1
        ju= j * nxny
c$dir no_recurrence
        do 20 i = jl, ju
          px(i) = x(i) - a(i-nxny,4) * px(i-nxny)
   20 continue
        call precd( px(jl), nxny, a(jl,2), a(jl,3), nx, left )
   30 continue
      rkr = ddot( n, px, 1, px, 1 )
      left = .false.
*
* Solving the upper triangular part of the preconditioning
*
      jl = (nz - 1) * nxny + 1
```

```
      call precd( px(jl), nxny, a(jl,2), a(jl,3), nx, left )
      do 50 j = nz-1, 1, -1
        jl = (j - 1) * nxny + 1
        ju = j * nxny
c$dir no_recurrence          do 40 i = jl, ju
          px(i) = px(i) - a(i,4) * px(i+nxny)
 40   continue
        call precd( px(jl), nxny, a(jl,2), a(jl,3), nx, left )
 50   continue
      return
      end
            subroutine precd( px, nxny, a2, a3, nx, left )
      implicit double precision (a-h,o-z)
      dimension px(*), a2(*), a3(*)
      logical left
*
* Diagonalwise computation in preconditioning over x,y-plane
* ([2]: Chapter 7.3.4)
*
      ny = nxny / nx
      if (.not.left) go to 15
      do 10 ii = 2, nx + ny - 1
        jjmin = max( 1, ii - nx + 1 )
        jjmax = min( ii, ny )
        is = (jjmin - 1) * (nx - 1) + ii
c$dir no_recurrence
        do 5 j = jjmin, jjmax
          px(is) = px(is) - a2(is-1) * px(is-1) - a3(is-nx) * px(is-nx)
          is = is + nx - 1
  5   continue
 10   continue
 10   continue
      return       *   15  continue
      do 30 ii = nx + ny - 1, 2, -1
        jjmin = max( 1, ii - nx + 1 )
        jjmax = min( ii, ny )
        is = (jjmin - 1) * (nx - 1) + ii
c$dir no_recurrence
        do 20 j = jjmin, jjmax
          px(is) = px(is) - a2(is) * px(is+1) - a3(is) * px(is+nx)
          is = is + nx - 1
 20   continue
 30   continue
      px(1) = px(1) - a2(1) * px(2) - a3(1) * px(nx+1)
      return
      end
```

C. Partly vectorized standard ICCG

The subroutine `prec3s` can be partly vectorized. This approach leads to code as in `prec3p` for which the compiler may detect that there is a recurrence left only in the first index (i.e., in the z-direction of the grid). The call to `prec3s` in the calling routine `iccg` should be changed accordingly in a call to `prec3p`.

The motivation for providing this version is the following. In some situations, when the matrix A has a different non-zero structure, vectorization by different ordering (as in `prec3d`) is not possible. In such cases partial vectorization might be appropriate. Comparing the performances of `prec3p` and `prec3s` then will give an idea of the effects of partial vectorization that one may expect in more general situations.

```
      subroutine prec3s( x, px, a2, a3, a4, nx, ny, nz, rkr )
      implicit double precision (a-h,o-z)
      dimension x(nx,ny,nz), px(nx,ny,nz), a2(nx,ny,nz), a3(nx,ny,nz), a4(nx,ny,nz)
      px(1,1,1) = x(1,1,1)
      do i = 2, nx
        px(i,1,1) = x(i,1,1) - a2(i-1,1,1) * px(i-1,1,1)
      enddo
      do j = 2, ny
        do i = 1, nx
          px(i,j,1) = x(i,j,1) - a2(i-1,j,1) * px(i-1,j,1) - a3(i,j-1,1) *
     &      px(i,j-1,1)
        enddo
      enddo
      do k = 2, nz
        do j = 1, ny
          do i = 1, nx
            px(i,j,k) = x(i,j,k) - a2(i-1,j,k) * px(i-1,j,k) - a3(i,j-1,k) *
     &      px(i,j-1,k) - a4(i,j,k-1) * px(i,j,k-1)
        enddo
        enddo
      enddo
*
* Lower triangular part has been solved
*
* Computation of (r,w), where w is the preconditioned residual
*
      rkr = ddot( nx * ny * nz, px, 1, px, 1 )
      do i = nx-1, 1, -1
        px(i,ny,nz) = px(i,ny,nz) - a2(i,ny,nz) * px(i+1,ny,nz)
      enddo
      do j = ny-1, 1, -1
        do i = nx, 1, -1
          px(i,j,nz) = px(i,j,nz) - a2(i,j,nz) * px(i+1,j,nz) - a3(i,j,nz) *
     &      px(i,j+1,nz)
```

```
        enddo
      enddo
      do k = nz-1, 1, -1
        do j = ny, 1, -1
          do i = nx, 1, -1
            px(i,j,k) = px(i,j,k) - a2(i,j,k) * px(i+1,j,k) - a3(i,j,k) *
     &      px(i,j+1,k) - a4(i,j,k) * px(i,j,k+1)
          enddo
        enddo
      enddo
*
* Diagonal and upper triangular part has been solved
*
      return
      end
```